Assignment 3

- Create a collection known as bag
- Your implementation must have all the Access Operations 1
- Extra credit for implementing Access Operations 2

Accessing A Collection

- Depends on the purpose and implementation of a collection, accesses to a collection may be very different.
  - We start with the simplest type of collection: bags.
- A **bag** is a random collection, aka, elements exhibit no particular positional relationship at all.
  - A bag here is similar to a physical bag into which things are placed.
  - items can be added, removed, accessed
  - no implied order to the items
  - duplicates allowed

Access Operations 1

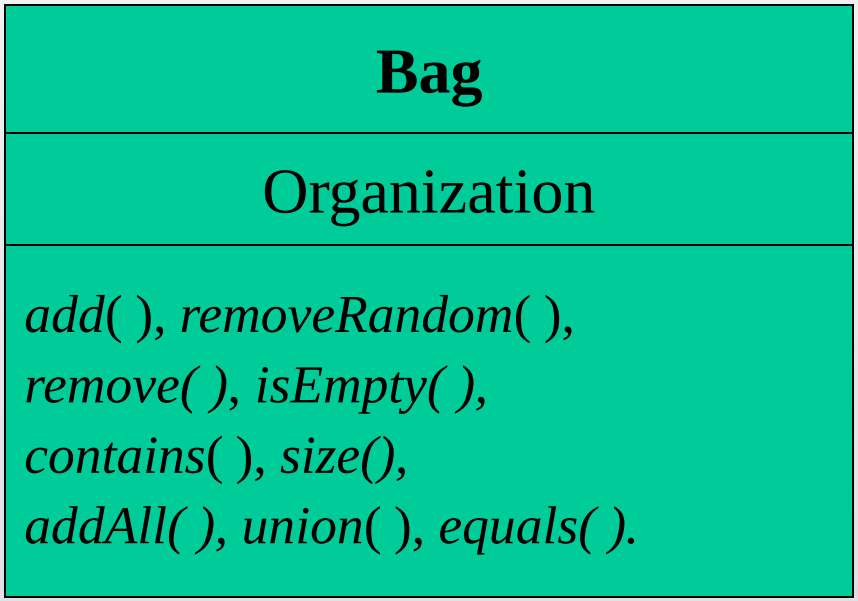
- *add*: adds an element to the bag.
- *remove*: removes a particular element form the bag.
- *removeRandom*: removes an element randomly from the bag.
- *isEmpty*: determines if the bag is empty.
- *contains*: determines if a particular element is in the bag.
- *size*: determines the number of elements in the bag.

Access Operations (2)

- *addAll:* adds the elements of one bag to another.
- *union*: combines the elements of two bags to create a third.
- *equals*: determines if two bags contains the same elements (namely whether they are the same as sets).

Implementation Issues

- We need to separate organization and operations (or interface).
  - The user of a bag needs to focus on how to use the bag without worrying about how it is organized internally.



Organization of Elements

- We use array (a linear collection) to represent the bag. This seems contradicting to the nature of the bag (a “random” collection).
  - When implementing a structure, choose programming constructs, which are available and easy to use.
  - We will analyze the complexities of bag various operations later, and we will see the array idea is not a mistake.
- Our running example is an integer array for illustrations; the idea applies to array of all data types, including objects.

Implementation Ideas, cont'd

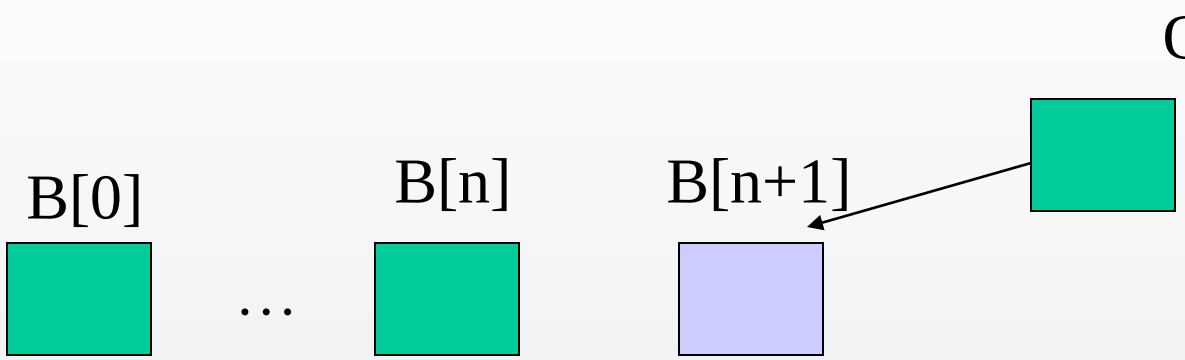
```
class Bag
{
        private object b[MAX];
        int count; // Number of elements
```

```
        public void add(object );
        public void removeRandom( );
        public void remove(object );
        public boolean isEmpty( );
        public boolean contains(object );
        public int size();
        public Bag addAll(Bag );
        public Bag union(Bag, Bag );
        public boolean equals(Bag );

} // End of class
```

- Note: You can separate organization and access/interface in different classes. You may use interface feature in Java to implement the access part.

Operation: add( )



Operation *add*( ). Need to update: *count++;*

Refinements:

If the bag is full, then a new element cannot be added; an exception must be caught.

Another option is to expand the capacity MAX.

Operation: *remove*( )

- To remove an object from the bag, first check whether the bag contains the element. So *contains()* can be used.
  - For writing *contains( )*, use linear search.
- If the object is not found, an exception should be caught. Otherwise perform the following steps.
  - Move the last element in the array down to fill the position of the found element.
  - *count--;*
- *removeRandom( )* is easier:
  - First generate a random position, then remove the object from that position like *remove( )* does.

Easiest Parts

isEmpty( ):

size( ):

Operations on Multiple Bags

- *addAll( ):* Use *for* loop to add elements from the first bag (from the the start) to the second bag (from the end). Again, checking should be done to avoid overflow.
- *union( )* of two bags is done using two *for* loops.
- *equals( )* can be done as follows:
  1. Check whether they have the same size (obvious, huh?)
  2. Then create a copy for each bag.
  3. Unite the two copies to form a larger bag.
  4. For every element in the larger, use *contain*( ) to check whether it is in both copies, if no, return false; if yes, remove the element from both bags (copies). Repeat the process until both copies are empty.

Complexity Analysis for One-bag Operations

- Assume the size of the bag is *n*.
  - *add*( ): O(1).
  - *contains*( ): O(n). It is basically the cost of linear searching.
  - *remove*( ): O(n)
  - *removeRandom*( ): O(1). No search is needed.
  - *size*( ): O(1)
  - *isEmpty*( ): O(1)

Complexity Analysis for Two-bag Operations

- Assume bag1 and bag2 are of sizes n and m, respectively.
- Add all of bag1 to bag2: $O(n)$.
- Union of bag1 and bag2: $O(n)+O(m)=O(n+m)$.
- The call bag1.equals(bag2): $O(1)+O(n)+O(n)+O(n+n)+2n*(O(n)+O(n))=O(n^2)$.